Jumping Into Smalltalk

I hear from many Java and C# programmers that they can't read Smalltalk code. Smalltalk is actually very easy to learn since it has very little syntax. The true power of Smalltalk lies in the environment and the class libraries.

This article is intended to give Java and C# programmers an extremely rapid plunge into Smalltalk. I'll do that by using side-by-side code comparisons.

Temporary Variables

Smalltalk doesn't require type declarations on any variables. Temporary variables are defined within vertical bars.

Java	Smalltalk

int a;
char b;
float c;
	| a b c |

Assignment

Smalltalk uses := for assignment.

Java	Smalltalk
a = 5;	a := 5

Messages

Smalltalk has three kinds of messages.

Type	Form	Parameters	Examples
unary	alphanumeric starting with a lowercase letter	0	squared

binary	punctuation marks	1	+
keyword	multiple colon terminated alphanumeric words	1 or more	do: between: and:

To pass one or more parameters, you would usually use a keyword message. Each parameter is preceded by a keyword. Smalltalk doesn't use brackets and commas to separate the parameters.

Examples

Java	**Smalltalk**
myAccount.getBalance();	myAccount getBalance
myAccount.setBalance (10);	myAccount setBalance: 10
myAccount.transfer (20, anotherAccount)	myAccount transfer: 20 to: anotherAccount
myAccount.equals (anotherAccount);	myAccount = anotherAccount

Order of operations is:

Unary (evaluate first)
Binary (evaluate second)
Keyword (evaluate last)

Within each priority level, evaluate left to right.

Java	**Smalltalk**
3 + 5 * 6 // answer: 33	3 + 5 * 6 "answer: 48"

Notice that in the Smalltalk version, this expression is actually two messages:

Message 1
  receiver: 3
  message: +
  parameter: 5

result: 8

Message 2
receiver: 8
message: *
parameter: 6

result: 48

Statements

Smalltalk uses a period (.) as a statement separator. You don't need a period on the last statement.

Java	Smalltalk

myAccount.deposit(20);
myAccount.transfer(20, anotherAccount);
	myAccount deposit: 20.
	myAccount transfer: 20 to: anotherAccount

Literals

In Smalltalk, integers, characters, strings, booleans, floats and doubles are all first class objects. Integers are infinite precision and automatically grow as needed without overflow. As such, there's no equivalent to char, byte, short, int, or long. They're all just integers.

Java	Smalltalk
5	5
01230	8r1230
0x7f	16r7f
<no equivalent>	3r21012 (you can use any base you like)
200L	<no equivalent>
2e-5	2e-5
2e-5d	2d-5
'h'	$h

'\u03A9'	Character value: 16r3A9
"hello"	'hello'
"can't"	'can''t'
{"a","b","c"}	#('a' 'b' 'c')
<no equivalent>	#($a 234 #hello)

Special Words

In Smalltalk, nil refers to a real object. It's an instance of the class UndefinedObject. The word true refers to an instance of the class True and false refers to an instance of the class False.

Java	Smalltalk
this	self
null	nil
true	true
false	false
super base (C#)	super

Returning From Methods

Java	Smalltalk
return value;	^value

Cascades

Smalltalk uses a semicolon (;) to separate multiple messages sent to the same object.

Java	Smalltalk

<no equivalent>	myAccount
	   deposit: 20;
	   transfer: 20 to: anotherAccount

Comments

Java	Smalltalk
/* comment */ // another comment	"comment"

Instance Creation

In Smalltalk, classes are real objects. To create an instance, just send new to the class. Methods for a class are called class methods (similar to Java static methods).

Java	Smalltalk
new Reservation();	Reservation new

Constructors

Smalltalk has no constructors. If you want to perform instance initialization, you can redefine the "new" class method to initialize the instance.

Java	Smalltalk

Reservation()
{
  startTime =
    new GregorianCalendar().getTime();
  endTime =
    new GregorianCalendar().getTime();
}
	Reservation class method:
	new
	  ^super new initialize
	
	Reservation instance method:
	initialize
	  startTime := Timestamp now.
	  endTime := Timestamp now

Methods

Java	Smalltalk

class Room {
 	
  void book (DateTime start, DateTime end)
  {
    reservations.add (
      new Reservation(start, end));
  }
 	
}
	bookFrom: startTime to: endTime
	  reservations add:
	    (Reservation from: startTime to: endTime)

Blocks

Smalltalk has an object called a block. It's an object that contains executable code. The closest thing Java has is an anonymous inner class. In C# 2.0, there are anonymous delegates that are similar.

To execute a block with no parameters, you send it a **value** message.

Smalltalk
| block | block := [3 + 4]. block value   "answer is 7"

Blocks can have parameters. Each block parameter declaration starts with a colon (:). A vertical bar (|) denotes the end of the parameter list and the start of the code for the block.

Smalltalk
| block | block := [:x :y | x * 2 + y]. block value: 5 value: 3   "answer is 13"

End of the Syntax

At this point, we've covered all of the syntax of Smalltalk. Everything else is part of the class library. Have you noticed anything missing? How about if-then-else or while loops? Smalltalk just uses blocks and ordinary message sends.

Control Structures

Smalltalk has no control structures like if built into the language. Instead, Smalltalk uses messages sent to the true or false objects.

Java	Smalltalk

if (tries > 5)
   return "Too many tries";
else
   return "Trying again";
	tries > 5
	   ifTrue: [^'Too many tries']
	   ifFalse: [^'Trying again']

Notice that the ^ returns from the method, not just the block.

Loops

Smalltalk uses blocks to do looping. Since blocks are just objects, we can send messages to them.

Java	Smalltalk

int tries = 0;
while (tries <= 5) {
   tryAgain();
   tries++;
   }
	| tries |
	tries := 0.
	[tries <= 5] whileTrue: [
	   self tryAgain.
	   tries := tries + 1]
	
	alternatively:
	
	5 timesRepeat: [self tryAgain]

Notice that timesRepeat: is a message understood by Integers. It simply evaluates the block the proper number of times.

Things left to learn

This concludes this quick introduction to Smalltalk. There are still many things to learn but there is no more syntax. Everything else is part of the class library.

If you'd like to learn more, download Smalltalk try it out. Wilf Lalonde has a good article on learning Smalltalk for Java and C++ programmers. For lots of Smalltalk resources including links to online Smalltalk books, visit the Why Smalltalk site. If you'd be interested in formal training contact Simberon and ask about Smalltalk training

Chris Rathman / ChrisRath@aol.com

```
"************************************************************************
 * Allowable characters:                                                *
 *    - a-z                                                             *
 *    - A-Z                                                             *
 *    - 0-9                                                             *
 *    - .+/\*~<>@%|&?                                                   *
 *    - blank, tab, cr, ff, lf                                          *
 *                                                                      *
 * Variables:                                                           *
 *    - variables must be declared before use                           *
 *    - shared vars must begin with uppercase                           *
 *    - local vars must begin with lowercase                            *
 *    - reserved names: nil, true, false, self, super, and Smalltalk    *
 *                                                                      *
 * Variable scope:                                                      *
 *    - Global: defined in Dictionary Smalltalk and accessible by all   *
 *        objects in system                                             *
 *    - Special: (reserved) Smalltalk, super, self, true, false, & nil  *
 *    - Method Temporary: local to a method                             *
 *    - Block Temporary: local to a block                               *
 *    - Pool: variables in a Dictionary object                          *
 *    - Method Parameters: automatic local vars created as a result of  *
 *        message call with params                                      *
 *    - Block Parameters: automatic local vars created as a result of   *
 *        value: message call                                           *
 *    - Class: shared with all instances of one class & its subclasses  *
 *    - Class Instance: unique to each instance of a class              *
 *    - Instance Variables: unique to each instance                     *
 ************************************************************************"
"Comments are enclosed in quotes"
"Period (.) is the statement seperator"

"************************************************************************
 * Transcript:                                                          *
 ************************************************************************"
Transcript clear.                                           "clear to transcript window"
Transcript show: 'Hello World'.                             "output string in transcript window"
Transcript nextPutAll: 'Hello World'.                       "output string in transcript window"
Transcript nextPut: $A.                                     "output character in transcript window"
Transcript space.                                           "output space character in transcript window"
Transcript tab.                                             "output tab character in transcript window"
Transcript cr.                                              "carriage return / linefeed"
'Hello' printOn: Transcript.                                "append print string into the window"
'Hello' storeOn: Transcript.                                "append store string into the window"
Transcript endEntry.                                        "flush the output buffer"

"************************************************************************
 * Assignment:                                                          *
 ************************************************************************"
| x y |
x _ 4.                                                      "assignment (Squeak) <-"
x := 5.                                                     "assignment"
x := y := z := 6.                                           "compound assignment"
x := (y := 6) + 1.
x := Object new.                                            "bind to allocated instance of a class"
x := 123 class.                                             "discover the object class"
x := Integer superclass.                                    "discover the superclass of a class"
x := Object allInstances.                                   "get an array of all instances of a class"
x := Integer allSuperclasses.                               "get all superclasses of a class"
x := 1.2 hash.                                              "hash value for object"
y := x copy.                                                "copy object"
y := x shallowCopy.                                         "copy object (not overridden)"
y := x deepCopy.                                            "copy object and instance vars"
y := x veryDeepCopy.                                        "complete tree copy using a dictionary"

"************************************************************************
 * Constants:                                                           *
 ************************************************************************"
| b |
b := true.                                                  "true constant"
b := false.                                                 "false constant"
x := nil.                                                   "nil object constant"
x := 1.                                                     "integer constants"
x := 3.14.                                                  "float constants"
x := 2e-2.                                                  "fractional constants"
x := 16r0F.                                                 "hex constant".
x := -1.                                                    "negative constants"
```

```
x := 'Hello'.                                                 "string constant"
x := 'I''m here'.                                             "single quote escape"
x := $A.                                                      "character constant"
x := $ .                                                      "character constant (space)"
x := #aSymbol.                                                "symbol constants"
x := #(3 2 1).                                                "array constants"
x := #('abc' 2 $a).                                           "mixing of types allowed"

"************************************************************************
 * Booleans:                                                            *
 ************************************************************************"
| b x y |
x := 1. y := 2.
b := (x = y).                                                 "equals"
b := (x ~= y).                                                "not equals"
b := (x == y).                                                "identical"
b := (x ~~ y).                                                "not identical"
b := (x > y).                                                 "greater than"
b := (x < y).                                                 "less than"
b := (x >= y).                                                "greater than or equal"
b := (x <= y).                                                "less than or equal"
b := b not.                                                   "boolean not"
b := (x < 5) & (y > 1).                                       "boolean and"
b := (x < 5) | (y > 1).                                       "boolean or"
b := (x < 5) and: [y > 1].                                    "boolean and (short-circuit)"
b := (x < 5) or: [y > 1].                                     "boolean or (short-circuit)"
b := (x < 5) eqv: (y > 1).                                    "test if both true or both false"
b := (x < 5) xor: (y > 1).                                    "test if one true and other false"
b := 5 between: 3 and: 12.                                    "between (inclusive)"
b := 123 isKindOf: Number.                                    "test if object is class or subclass of"
b := 123 isMemberOf: SmallInteger.                            "test if object is type of class"
b := 123 respondsTo: sqrt.                                    "test if object responds to message"
b := x isNil.                                                 "test if object is nil"
b := x isZero.                                                "test if number is zero"
b := x positive.                                              "test if number is positive"
b := x strictlyPositive.                                      "test if number is greater than zero"
b := x negative.                                              "test if number is negative"
b := x even.                                                  "test if number is even"
b := x odd.                                                   "test if number is odd"
b := x isLiteral.                                             "test if literal constant"
b := x isInteger.                                             "test if object is integer"
b := x isFloat.                                               "test if object is float"
b := x isNumber.                                              "test if object is number"
b := $A isUppercase.                                          "test if upper case character"
b := $A isLowercase.                                          "test if lower case character"

"************************************************************************
 * Arithmetic expressions:                                              *
 ************************************************************************"
| x |
x := 6 + 3.                                                   "addition"
x := 6 - 3.                                                   "subtraction"
x := 6 * 3.                                                   "multiplication"
x := 1 + 2 * 3.                                               "evaluation always left to right (1 + 2) * 3"
x := 5 / 3.                                                   "division with fractional result"
x := 5.0 / 3.0.                                               "division with float result"
x := 5.0 // 3.0.                                              "integer divide"
x := 5.0 \\ 3.0.                                              "integer remainder"
x := -5.                                                      "unary minus"
x := 5 sign.                                                  "numeric sign (1, -1 or 0)"
x := 5 negated.                                               "negate receiver"
x := 1.2 integerPart.                                         "integer part of number (1.0)"
x := 1.2 fractionPart.                                        "fractional part of number (0.2)"
x := 5 reciprocal.                                            "reciprocal function"
x := 6 * 3.1.                                                 "auto convert to float"
x := 5 squared.                                               "square function"
x := 25 sqrt.                                                 "square root"
x := 5 raisedTo: 2.                                           "power function"
x := 5 raisedToInteger: 2.                                    "power function with integer"
x := 5 exp.                                                   "exponential"
x := -5 abs.                                                  "absolute value"
x := 3.99 rounded.                                            "round"
x := 3.99 truncated.                                          "truncate"
x := 3.99 roundTo: 1.                                         "round to specified decimal places"
x := 3.99 truncateTo: 1.                                      "truncate to specified decimal places"
x := 3.99 floor.                                              "truncate"
x := 3.99 ceiling.                                            "round up"
x := 5 factorial.                                             "factorial"
x := -5 quo: 3.                                               "integer divide rounded toward zero"
x := -5 rem: 3.                                               "integer remainder rounded toward zero"
x := 28 gcd: 12.                                              "greatest common denominator"
```

```
x := 28 lcm: 12.                                            "least common multiple"
x := 100 ln.                                                "natural logarithm"
x := 100 log.                                               "base 10 logarithm"
x := 100 log: 10.                                           "logarithm with specified base"
x := 100 floorLog: 10.                                      "floor of the log"
x := 180 degreesToRadians.                                  "convert degrees to radians"
x := 3.14 radiansToDegrees.                                 "convert radians to degrees"
x := 0.7 sin.                                               "sine"
x := 0.7 cos.                                               "cosine"
x := 0.7 tan.                                               "tangent"
x := 0.7 arcSin.                                            "arcsine"
x := 0.7 arcCos.                                            "arccosine"
x := 0.7 arcTan.                                            "arctangent"
x := 10 max: 20.                                            "get maximum of two numbers"
x := 10 min: 20.                                            "get minimum of two numbers"
x := Float pi.                                              "pi"
x := Float e.                                               "exp constant"
x := Float infinity.                                        "infinity"
x := Float nan.                                             "not-a-number"
x := Random new next; yourself. x next.                     "random number stream (0.0 to 1.0)
x := 100 atRandom.                                          "quick random number"

"************************************************************************
 * Bitwise Manipulation:                                                *
 ************************************************************************"
| b x |
x := 16rFF bitAnd: 16r0F.                                   "and bits"
x := 16rF0 bitOr: 16r0F.                                    "or bits"
x := 16rFF bitXor: 16r0F.                                   "xor bits"
x := 16rFF bitInvert.                                       "invert bits"
x := 16r0F bitShift: 4.                                     "left shift"
x := 16rF0 bitShift: -4.                                    "right shift"
"x := 16r80 bitAt: 7."                                      "bit at position (0|1) [!Squeak]"
x := 16r80 highbit.                                         "position of highest bit set"
b := 16rFF allMask: 16r0F.                                  "test if all bits set in mask set in receiver"
b := 16rFF anyMask: 16r0F.                                  "test if any bits set in mask set in receiver"
b := 16rFF noMask: 16r0F.                                   "test if all bits set in mask clear in receiver"

"************************************************************************
 * Conversion:                                                          *
 ************************************************************************"
| x |
x := 3.99 asInteger.                                        "convert number to integer (truncates in Squeak)"
x := 3.99 asFraction.                                       "convert number to fraction"
x := 3 asFloat.                                             "convert number to float"
x := 65 asCharacter.                                        "convert integer to character"
x := $A asciiValue.                                         "convert character to integer"
x := 3.99 printString.                                      "convert object to string via printOn:"
x := 3.99 storeString.                                      "convert object to string via storeOn:"
x := 15 radix: 16.                                          "convert to string in given base"
x := 15 printStringBase: 16.
x := 15 storeStringBase: 16.

"************************************************************************
 * Blocks:                                                              *
 *    - blocks are objects and may be assigned to a variable            *
 *    - value is last expression evaluated unless explicit return       *
 *    - blocks may be nested                                            *
 *    - specification [ arguments | | localvars | expressions ]         *
 *    - Squeak does not currently support localvars in blocks           *
 *    - max of three arguments allowed                                  *
 *    - ^expression terminates block & method (exits all nested blocks) *
 *    - blocks intended for long term storage should not contain ^      *
 ************************************************************************"
| x y z |
x := [ y := 1. z := 2. ]. x value.                          "simple block usage"
x := [ :argOne :argTwo |   argOne, ' and ' , argTwo.].      "set up block with argument passing"
Transcript show: (x value: 'First' value: 'Second'); cr.    "use block with argument passing"
"x := [ | z | z := 1.].                                      localvars not available in squeak blocks"

"************************************************************************
 * Method calls:                                                        *
 *    - unary methods are messages with no arguments                    *
 *    - binary methods                                                  *
 *    - keyword methods are messages with selectors including colons    *
 *                                                                      *
 * standard categories/protocols:                                       *
 *    - initialize-release    (methods called for new instance)         *
 *    - accessing             (get/set methods)                         *
 *    - testing               (boolean tests - is)                      *
 *    - comparing             (boolean tests with parameter             *
  *    - displaying           (gui related methods)                    *
  *    - printing             (methods for printing)                   *
  *    - updating             (receive notification of changes)        *
  *    - private              (methods private to class)               *
  *    - instance-creation    (class methods for creating instance)    *
  *********************************************************************"
| x |
x := 2 sqrt.                                               "unary message"
x := 2 raisedTo: 10.                                       "keyword message"
x := 194 * 9.                                              "binary message"
Transcript show: (194 * 9) printString; cr.                "combination (chaining)"
x := 2 perform: #sqrt.                                     "indirect method invocation"
Transcript                                                 "Cascading - send multiple messages to receiver"
   show: 'hello ';
   show: 'world';
   cr.
x := 3 + 2; * 100.                                         "result=300. Sends message to same receiver (3)"

"*********************************************************************
 * Conditional Statements:                                           *
 *********************************************************************"
| x |
x > 10 ifTrue: [Transcript show: 'ifTrue'; cr].            "if then"
x > 10 ifFalse: [Transcript show: 'ifFalse'; cr].          "if else"
x > 10                                                     "if then else"
   ifTrue: [Transcript show: 'ifTrue'; cr]
   ifFalse: [Transcript show: 'ifFalse'; cr].
x > 10                                                     "if else then"
   ifFalse: [Transcript show: 'ifFalse'; cr]
   ifTrue: [Transcript show: 'ifTrue'; cr].
Transcript
   show:
      (x > 10
         ifTrue: ['ifTrue']
         ifFalse: ['ifFalse']);
   cr.
Transcript                                                 "nested if then else"
   show:
      (x > 10
         ifTrue: [x > 5
            ifTrue: ['A']
            ifFalse: ['B']]
         ifFalse: ['C']);
   cr.
switch := Dictionary new.                                  "switch functionality"
switch at: $A put: [Transcript show: 'Case A'; cr].
switch at: $B put: [Transcript show: 'Case B'; cr].
switch at: $C put: [Transcript show: 'Case C'; cr].
result := (switch at: $B) value.

"*********************************************************************
 * Iteration statements:                                             *
 *********************************************************************"
| x y |
x := 4. y := 1.
[x > 0] whileTrue: [x := x - 1. y := y * 2].               "while true loop"
[x >= 4] whileFalse: [x := x + 1. y := y * 2].             "while false loop"
x timesRepeat: [y := y * 2].                               "times repear loop (i := 1 to x)"
1 to: x do: [:a | y := y * 2].                             "for loop"
1 to: x by: 2 do: [:a | y := y / 2].                       "for loop with specified increment"
#(5 4 3) do: [:a | x := x + a].                            "iterate over array elements"

"*********************************************************************
 * Character:                                                        *
 *********************************************************************"
| x y |
x := $A.                                                   "character assignment"
y := x isLowercase.                                        "test if lower case"
y := x isUppercase.                                        "test if upper case"
y := x isLetter.                                           "test if letter"
y := x isDigit.                                            "test if digit"
y := x isAlphaNumeric.                                     "test if alphanumeric"
y := x isSeparator.                                        "test if seperator char"
y := x isVowel.                                            "test if vowel"
y := x digitValue.                                         "convert to numeric digit value"
y := x asLowercase.                                        "convert to lower case"
y := x asUppercase.                                        "convert to upper case"
y := x asciiValue.                                         "convert to numeric ascii value"
y := x asString.                                           "convert to string"
b := $A <= $B.                                             "comparison"
y := $A max: $B.
```

```
"************************************************************************
 * Symbol:                                                              *
 ************************************************************************"
| b x y |
x := #Hello.                                            "symbol assignment"
y := 'String', 'Concatenation'.                         "symbol concatenation (result is string)"
b := x isEmpty.                                         "test if symbol is empty"
y := x size.                                            "string size"
y := x at: 2.                                           "char at location"
y := x copyFrom: 2 to: 4.                               "substring"
y := x indexOf: $e ifAbsent: [0].                       "first position of character within string"
x do: [:a | Transcript show: a printString; cr].         "iterate over the string"
b := x conform: [:a | (a >= $a) & (a <= $z)].           "test if all elements meet condition"
y := x select: [:a | a > $a].                           "return all elements that meet condition"
y := x asString.                                        "convert symbol to string"
y := x asText.                                          "convert symbol to text"
y := x asArray.                                         "convert symbol to array"
y := x asOrderedCollection.                             "convert symbol to ordered collection"
y := x asSortedCollection.                              "convert symbol to sorted collection"
y := x asBag.                                           "convert symbol to bag collection"
y := x asSet.                                           "convert symbol to set collection"

"************************************************************************
 * String:                                                              *
 ************************************************************************"
| b x y |
x := 'This is a string'.                                "string assignment"
x := 'String', 'Concatenation'.                         "string concatenation"
b := x isEmpty.                                         "test if string is empty"
y := x size.                                            "string size"
y := x at: 2.                                           "char at location"
y := x copyFrom: 2 to: 4.                               "substring"
y := x indexOf: $a ifAbsent: [0].                       "first position of character within string"
x := String new: 4.                                     "allocate string object"
x                                                       "set string elements"
   at: 1 put: $a;
   at: 2 put: $b;
   at: 3 put: $c;
   at: 4 put: $e.
x := String with: $a with: $b with: $c with: $d.        "set up to 4 elements at a time"
x do: [:a | Transcript show: a printString; cr].         "iterate over the string"
b := x conform: [:a | (a >= $a) & (a <= $z)].           "test if all elements meet condition"
y := x select: [:a | a > $a].                           "return all elements that meet condition"
y := x asSymbol.                                        "convert string to symbol"
y := x asArray.                                         "convert string to array"
x := 'ABCD' asByteArray.                                "convert string to byte array"
y := x asOrderedCollection.                             "convert string to ordered collection"
y := x asSortedCollection.                              "convert string to sorted collection"
y := x asBag.                                           "convert string to bag collection"
y := x asSet.                                           "convert string to set collection"
y := x shuffled.                                        "randomly shuffle string"

"************************************************************************
 * Array:         Fixed length collection                               *
 * ByteArray:     Array limited to byte elements (0-255)                *
 * WordArray:     Array limited to word elements (0-2^32)               *
 ************************************************************************"
| b x y sum max |
x := #(4 3 2 1).                                        "constant array"
x := Array with: 5 with: 4 with: 3 with: 2.             "create array with up to 4 elements"
x := Array new: 4.                                      "allocate an array with specified size"
x                                                       "set array elements"
   at: 1 put: 5;
   at: 2 put: 4;
   at: 3 put: 3;
   at: 4 put: 2.
b := x isEmpty.                                         "test if array is empty"
y := x size.                                            "array size"
y := x at: 4.                                           "get array element at index"
b := x includes: 3.                                     "test if element is in array"
y := x copyFrom: 2 to: 4.                               "subarray"
y := x indexOf: 3 ifAbsent: [0].                        "first position of element within array"
y := x occurrencesOf: 3.                                "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].         "iterate over the array"
b := x conform: [:a | (a >= 1) & (a <= 4)].             "test if all elements meet condition"
y := x select: [:a | a > 2].                            "return collection of elements that pass test"
y := x reject: [:a | a < 2].                            "return collection of elements that fail test"
y := x collect: [:a | a + a].                           "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                 "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.             "sum array elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum array elements"
sum := x inject: 0 into: [:a :c | a + c].                   "sum array elements"
max := x inject: 0 into: [:a :c | (a > c)                   "find max element in array"
   ifTrue: [a]
   ifFalse: [c]].
y := x shuffled.                                            "randomly shuffle collection"
y := x asArray.                                             "convert to array"
"y := x asByteArray."                                       "note: this instruction not available on Squeak"
y := x asWordArray.                                         "convert to word array"
y := x asOrderedCollection.                                 "convert to ordered collection"
y := x asSortedCollection.                                  "convert to sorted collection"
y := x asBag.                                               "convert to bag collection"
y := x asSet.                                               "convert to set collection"

"************************************************************************
 * OrderedCollection: acts like an expandable array                     *
 ************************************************************************"
| b x y sum max |
x := OrderedCollection with: 4 with: 3 with: 2 with: 1.     "create collection with up to 4 elements"
x := OrderedCollection new.                                 "allocate collection"
x add: 3; add: 2; add: 1; add: 4; yourself.                 "add element to collection"
y := x addFirst: 5.                                         "add element at beginning of collection"
y := x removeFirst.                                         "remove first element in collection"
y := x addLast: 6.                                          "add element at end of collection"
y := x removeLast.                                          "remove last element in collection"
y := x addAll: #(7 8 9).                                    "add multiple elements to collection"
y := x removeAll: #(7 8 9).                                 "remove multiple elements from collection"
x at: 2 put: 3.                                             "set element at index"
y := x remove: 5 ifAbsent: [].                              "remove element from collection"
b := x isEmpty.                                             "test if empty"
y := x size.                                                "number of elements"
y := x at: 2.                                               "retrieve element at index"
y := x first.                                               "retrieve first element in collection"
y := x last.                                                "retrieve last element in collection"
b := x includes: 5.                                         "test if element is in collection"
y := x copyFrom: 2 to: 3.                                   "subcollection"
y := x indexOf: 3 ifAbsent: [0].                            "first position of element within collection"
y := x occurrencesOf: 3.                                    "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].            "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].                 "test if all elements meet condition"
y := x select: [:a | a > 2].                                "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                "return collection of elements that fail test"
y := x collect: [:a | a + a].                               "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                     "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                 "sum elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                   "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                   "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x shuffled.                                            "randomly shuffle collection"
y := x asArray.                                             "convert to array"
y := x asOrderedCollection.                                 "convert to ordered collection"
y := x asSortedCollection.                                  "convert to sorted collection"
y := x asBag.                                               "convert to bag collection"
y := x asSet.                                               "convert to set collection"

"************************************************************************
 * SortedCollection:    like OrderedCollection except order of elements *
 *                        determined by sorting criteria                *
 ************************************************************************"
| b x y sum max |
x := SortedCollection with: 4 with: 3 with: 2 with: 1.      "create collection with up to 4 elements"
x := SortedCollection new.                                  "allocate collection"
x := SortedCollection sortBlock: [:a :c | a > c].           "set sort criteria"
x add: 3; add: 2; add: 1; add: 4; yourself.                 "add element to collection"
y := x addFirst: 5.                                         "add element at beginning of collection"
y := x removeFirst.                                         "remove first element in collection"
y := x addLast: 6.                                          "add element at end of collection"
y := x removeLast.                                          "remove last element in collection"
y := x addAll: #(7 8 9).                                    "add multiple elements to collection"
y := x removeAll: #(7 8 9).                                 "remove multiple elements from collection"
y := x remove: 5 ifAbsent: [].                              "remove element from collection"
b := x isEmpty.                                             "test if empty"
y := x size.                                                "number of elements"
y := x at: 2.                                               "retrieve element at index"
y := x first.                                               "retrieve first element in collection"
y := x last.                                                "retrieve last element in collection"
b := x includes: 4.                                         "test if element is in collection"
y := x copyFrom: 2 to: 3.                                   "subcollection"
y := x indexOf: 3 ifAbsent: [0].                            "first position of element within collection"
```

```
y := x occurrencesOf: 3.                                      "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].              "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].                   "test if all elements meet condition"
y := x select: [:a | a > 2].                                  "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                  "return collection of elements that fail test"
y := x collect: [:a | a + a].                                 "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                       "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                   "sum elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                     "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                     "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                               "convert to array"
y := x asOrderedCollection.                                   "convert to ordered collection"
y := x asSortedCollection.                                    "convert to sorted collection"
y := x asBag.                                                 "convert to bag collection"
y := x asSet.                                                 "convert to set collection"

"************************************************************************
 * Bag:        like OrderedCollection except elements are in no         *
 *                particular order                                      *
 ************************************************************************"
| b x y sum max |
x := Bag with: 4 with: 3 with: 2 with: 1.                     "create collection with up to 4 elements"
x := Bag new.                                                 "allocate collection"
x add: 4; add: 3; add: 1; add: 2; yourself.                   "add element to collection"
x add: 3 withOccurrences: 2.                                  "add multiple copies to collection"
y := x addAll: #(7 8 9).                                      "add multiple elements to collection"
y := x removeAll: #(7 8 9).                                   "remove multiple elements from collection"
y := x remove: 4 ifAbsent: [].                                "remove element from collection"
b := x isEmpty.                                               "test if empty"
y := x size.                                                  "number of elements"
b := x includes: 3.                                           "test if element is in collection"
y := x occurrencesOf: 3.                                      "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].              "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].                   "test if all elements meet condition"
y := x select: [:a | a > 2].                                  "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                  "return collection of elements that fail test"
y := x collect: [:a | a + a].                                 "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                       "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                   "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                     "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                     "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asOrderedCollection.                                   "convert to ordered collection"
y := x asSortedCollection.                                    "convert to sorted collection"
y := x asBag.                                                 "convert to bag collection"
y := x asSet.                                                 "convert to set collection"

"************************************************************************
 * Set:           like Bag except duplicates not allowed                *
 * IdentitySet:   uses identity test (== rather than =)                 *
 ************************************************************************"
| b x y sum max |
x := Set with: 4 with: 3 with: 2 with: 1.                     "create collection with up to 4 elements"
x := Set new.                                                 "allocate collection"
x add: 4; add: 3; add: 1; add: 2; yourself.                   "add element to collection"
y := x addAll: #(7 8 9).                                      "add multiple elements to collection"
y := x removeAll: #(7 8 9).                                   "remove multiple elements from collection"
y := x remove: 4 ifAbsent: [].                                "remove element from collection"
b := x isEmpty.                                               "test if empty"
y := x size.                                                  "number of elements"
x includes: 4.                                                "test if element is in collection"
x do: [:a | Transcript show: a printString; cr].              "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].                   "test if all elements meet condition"
y := x select: [:a | a > 2].                                  "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                  "return collection of elements that fail test"
y := x collect: [:a | a + a].                                 "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                       "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                   "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                     "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                     "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                               "convert to array"
y := x asOrderedCollection.                                   "convert to ordered collection"
y := x asSortedCollection.                                    "convert to sorted collection"
y := x asBag.                                                 "convert to bag collection"
y := x asSet.                                                 "convert to set collection"
```

```
"************************************************************************
 * Interval:                                                            *
 ************************************************************************"
| b x y sum max |
x := Interval from: 5 to: 10.                               "create interval object"
x := 5 to: 10.
x := Interval from: 5 to: 10 by: 2.                         "create interval object with specified increment"
x := 5 to: 10 by: 2.
b := x isEmpty.                                             "test if empty"
y := x size.                                                "number of elements"
x includes: 9.                                              "test if element is in collection"
x do: [:k | Transcript show: k printString; cr].            "iterate over interval"
b := x conform: [:a | (a >= 1) & (a <= 4)].                 "test if all elements meet condition"
y := x select: [:a | a > 7].                                "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                "return collection of elements that fail test"
y := x collect: [:a | a + a].                               "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                     "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                 "sum elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                   "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                   "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                             "convert to array"
y := x asOrderedCollection.                                 "convert to ordered collection"
y := x asSortedCollection.                                  "convert to sorted collection"
y := x asBag.                                               "convert to bag collection"
y := x asSet.                                               "convert to set collection"

"************************************************************************
 * Associations:                                                        *
 ************************************************************************"
| x y |
x := #myVar->'hello'.
y := x key.
y := x value.

"************************************************************************
 * Dictionary:                                                          *
 * IdentityDictionary:   uses identity test (== rather than =)          *
 ************************************************************************"
| b x y |
x := Dictionary new.                                        "allocate collection"
x add: #a->4; add: #b->3; add: #c->1; add: #d->2; yourself. "add element to collection"
x at: #e put: 3.                                            "set element at index"
b := x isEmpty.                                             "test if empty"
y := x size.                                                "number of elements"
y := x at: #a ifAbsent: [].                                 "retrieve element at index"
y := x keyAtValue: 3 ifAbsent: [].                          "retrieve key for given value with error block"
y := x removeKey: #e ifAbsent: [].                          "remove element from collection"
b := x includes: 3.                                         "test if element is in values collection"
b := x includesKey: #a.                                     "test if element is in keys collection"
y := x occurrencesOf: 3.                                    "number of times object in collection"
y := x keys.                                                "set of keys"
y := x values.                                              "bag of values"
x do: [:a | Transcript show: a printString; cr].            "iterate over the values collection"
x keysDo: [:a | Transcript show: a printString; cr].        "iterate over the keys collection"
x associationsDo: [:a | Transcript show: a printString; cr]."iterate over the associations"
x keysAndValuesDo: [:aKey :aValue | Transcript              "iterate over keys and values"
   show: aKey printString; space;
   show: aValue printString; cr].
b := x conform: [:a | (a >= 1) & (a <= 4)].                 "test if all elements meet condition"
y := x select: [:a | a > 2].                                "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                "return collection of elements that fail test"
y := x collect: [:a | a + a].                               "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                     "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                 "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                   "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                   "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                             "convert to array"
y := x asOrderedCollection.                                 "convert to ordered collection"
y := x asSortedCollection.                                  "convert to sorted collection"
y := x asBag.                                               "convert to bag collection"
y := x asSet.                                               "convert to set collection"

Smalltalk at: #CMRGlobal put: 'CMR entry'.                  "put global in Smalltalk Dictionary"
x := Smalltalk at: #CMRGlobal.                              "read global from Smalltalk Dictionary"
Transcript show: (CMRGlobal printString).                   "entries are directly accessible by name"
```

```
Smalltalk keys do: [ :k |                                  "print out all classes"
   ((Smalltalk at: k) isKindOf: Class)
      ifFalse: [Transcript show: k printString; cr]].
Smalltalk at: #CMRDictionary put: (Dictionary new).        "set up user defined dictionary"
CMRDictionary at: #MyVar1 put: 'hello1'.                   "put entry in dictionary"
CMRDictionary add: #MyVar2->'hello2'.                      "add entry to dictionary use key->value combo"
CMRDictionary size.                                        "dictionary size"
CMRDictionary keys do: [ :k |                              "print out keys in dictionary"
   Transcript show: k printString; cr].
CMRDictionary values do: [ :k |                            "print out values in dictionary"
   Transcript show: k printString; cr].
CMRDictionary keysAndValuesDo: [:aKey :aValue |            "print out keys and values"
   Transcript
      show: aKey printString;
      space;
      show: aValue printString;
      cr].
CMRDictionary associationsDo: [:aKeyValue |                "another iterator for printing key values"
   Transcript show: aKeyValue printString; cr].
Smalltalk removeKey: #CMRGlobal ifAbsent: [].              "remove entry from Smalltalk dictionary"
Smalltalk removeKey: #CMRDictionary ifAbsent: [].          "remove user dictionary from Smalltalk dictionary"

"************************************************************************
 * Internal Stream:                                                     *
 ************************************************************************"
| b x ios |
ios := ReadStream on: 'Hello read stream'.
ios := ReadStream on: 'Hello read stream' from: 1 to: 5.
[(x := ios nextLine) notNil]
   whileTrue: [Transcript show: x; cr].
ios position: 3.
ios position.
x := ios next.
x := ios peek.
x := ios contents.
b := ios atEnd.

ios := ReadWriteStream on: 'Hello read stream'.
ios := ReadWriteStream on: 'Hello read stream' from: 1 to: 5.
ios := ReadWriteStream with: 'Hello read stream'.
ios := ReadWriteStream with: 'Hello read stream' from: 1 to: 10.
ios position: 0.
[(x := ios nextLine) notNil]
   whileTrue: [Transcript show: x; cr].
ios position: 6.
ios position.
ios nextPutAll: 'Chris'.
x := ios next.
x := ios peek.
x := ios contents.
b := ios atEnd.

"************************************************************************
 * FileStream:                                                          *
 ************************************************************************"
| b x ios |
ios := FileStream newFileNamed: 'ios.txt'.
ios nextPut: $H; cr.
ios nextPutAll: 'Hello File'; cr.
'Hello File' printOn: ios.
'Hello File' storeOn: ios.
ios close.

ios := FileStream oldFileNamed: 'ios.txt'.
[(x := ios nextLine) notNil]
   whileTrue: [Transcript show: x; cr].
ios position: 3.
x := ios position.
x := ios next.
x := ios peek.
b := ios atEnd.
ios close.

"************************************************************************
 * Date:                                                                *
 ************************************************************************"
| x y |
x := Date today.                                           "create date for today"
x := Date dateAndTimeNow.                                  "create date from current time/date"
x := Date readFromString: '01/02/1999'.                    "create date from formatted string"
x := Date newDay: 12 month: #July year: 1999               "create date from parts"
```

```
x := Date fromDays: 36000.                          "create date from elapsed days since 1/1/1901"
y := Date dayOfWeek: #Monday.                       "day of week as int (1-7)"
y := Date indexOfMonth: #January.                   "month of year as int (1-12)"
y := Date daysInMonth: 2 forYear: 1996.             "day of month as int (1-31)"
y := Date daysInYear: 1996.                         "days in year (365|366)"
y := Date nameOfDay: 1                              "weekday name (#Monday,...)"
y := Date nameOfMonth: 1.                           "month name (#January,...)"
y := Date leapYear: 1996.                           "1 if leap year; 0 if not leap year"
y := x weekday.                                     "day of week (#Monday,...)"
y := x previous: #Monday.                           "date for previous day of week"
y := x dayOfMonth.                                  "day of month (1-31)"
y := x day.                                         "day of year (1-366)"
y := x firstDayOfMonth.                             "day of year for first day of month"
y := x monthName.                                   "month of year (#January,...)"
y := x monthIndex.                                  "month of year (1-12)"
y := x daysInMonth.                                 "days in month (1-31)"
y := x year.                                        "year (19xx)"
y := x daysInYear.                                  "days in year (365|366)"
y := x daysLeftInYear.                              "days left in year (364|365)"
y := x asSeconds.                                   "seconds elapsed since 1/1/1901"
y := x addDays: 10.                                 "add days to date object"
y := x subtractDays: 10.                            "subtract days to date object"
y := x subtractDate: (Date today).                  "subtract date (result in days)"
y := x printFormat: #(2 1 3 $/ 1 1).                "print formatted date"
b := (x <= Date today).                             "comparison"

"************************************************************************
 * Time:                                                                *
 ************************************************************************"
| x y |
x := Time now.                                      "create time from current time"
x := Time dateAndTimeNow.                           "create time from current time/date"
x := Time readFromString: '3:47:26 pm'.             "create time from formatted string"
x := Time fromSeconds: (60 * 60 * 4).               "create time from elapsed time from midnight"
y := Time millisecondClockValue.                    "milliseconds since midnight"
y := Time totalSeconds.                             "total seconds since 1/1/1901"
y := x seconds.                                     "seconds past minute (0-59)"
y := x minutes.                                     "minutes past hour (0-59)"
y := x hours.                                       "hours past midnight (0-23)"
y := x addTime: (Time now).                         "add time to time object"
y := x subtractTime: (Time now).                    "subtract time to time object"
y := x asSeconds.                                   "convert time to seconds"
x := Time millisecondsToRun: [                      "timing facility"
   1 to: 1000 do: [:index | y := 3.14 * index]].
b := (x <= Time now).                               "comparison"

"************************************************************************
 * Point:                                                               *
 ************************************************************************"
| x y |
x := 200@100.                                       "obtain a new point"
y := x x.                                           "x coordinate"
y := x y.                                           "y coordinate"
x := 200@100 negated.                               "negates x and y"
x := (-200@-100) abs.                               "absolute value of x and y"
x := (200.5@100.5) rounded.                         "round x and y"
x := (200.5@100.5) truncated.                       "truncate x and y"
x := 200@100 + 100.                                 "add scale to both x and y"
x := 200@100 - 100.                                 "subtract scale from both x and y"
x := 200@100 * 2.                                   "multiply x and y by scale"
x := 200@100 / 2.                                   "divide x and y by scale"
x := 200@100 // 2.                                  "divide x and y by scale"
x := 200@100 \\ 3.                                  "remainder of x and y by scale"
x := 200@100 + 50@25.                               "add points"
x := 200@100 - 50@25.                               "subtract points"
x := 200@100 * 3@4.                                 "multiply points"
x := 200@100 // 3@4.                                "divide points"
x := 200@100 max: 50@200.                           "max x and y"
x := 200@100 min: 50@200.                           "min x and y"
x := 20@5 dotProduct: 10@2.                         "sum of product (x1*x2 + y1*y2)"

"************************************************************************
 * Rectangle:                                                           *
 ************************************************************************"
Rectangle fromUser.

"************************************************************************
 * Pen:                                                                 *
 ************************************************************************"
| myPen |
Display restoreAfter: [
   Display fillWhite.

myPen := Pen new.                                             "get graphic pen"
myPen squareNib: 1.
myPen color: (Color blue).                                    "set pen color"
myPen home.                                                   "position pen at center of display"
myPen up.                                                     "makes nib unable to draw"
myPen down.                                                   "enable the nib to draw"
myPen north.                                                  "points direction towards top"
myPen turn: -180.                                             "add specified degrees to direction"
myPen direction.                                              "get current angle of pen"
myPen go: 50.                                                 "move pen specified number of pixels"
myPen location.                                               "get the pen position"
myPen goto: 200@200.                                          "move to specified point"
myPen place: 250@250.                                         "move to specified point without drawing"
myPen print: 'Hello World' withFont: (TextStyle default fontAt: 1).
Display extent.                                               "get display width@height"
Display width.                                                "get display width"
Display height.                                               "get display height"

].

"************************************************************************
 * Dynamic Message Calling/Compiling:                                   *
 ************************************************************************"
| receiver message result argument keyword1 keyword2 argument1 argument2 |
"unary message"
receiver := 5.
message := 'factorial' asSymbol.
result := receiver perform: message.
result := Compiler evaluate: ((receiver storeString), ' ', message).
result := (Message new setSelector: message arguments: #()) sentTo: receiver.

"binary message"
receiver := 1.
message := '+' asSymbol.
argument := 2.
result := receiver perform: message withArguments: (Array with: argument).
result := Compiler evaluate: ((receiver storeString), ' ', message, ' ', (argument storeString)).
result := (Message new setSelector: message arguments: (Array with: argument)) sentTo: receiver.

"keyword messages"
receiver := 12.
keyword1 := 'between:' asSymbol.
keyword2 := 'and:' asSymbol.
argument1 := 10.
argument2 := 20.
result := receiver
   perform: (keyword1, keyword2) asSymbol
   withArguments: (Array with: argument1 with: argument2).
result := Compiler evaluate:
   ((receiver storeString), ' ', keyword1, (argument1 storeString) , ' ', keyword2, (argument2 storeString)).
result := (Message
   new
      setSelector: (keyword1, keyword2) asSymbol
      arguments: (Array with: argument1 with: argument2))
   sentTo: receiver.

"************************************************************************
 * class/meta-class:                                                    *
 ************************************************************************"
| b x |
x := String name.                                             "class name"
x := String category.                                         "organization category"
x := String comment.                                          "class comment"
x := String kindOfSubclass.                                   "subclass type - subclass: variableSubclass, etc"
x := String definition.                                       "class definition"
x := String instVarNames.                                     "immediate instance variable names"
x := String allInstVarNames.                                  "accumulated instance variable names"
x := String classVarNames.                                    "immediate class variable names"
x := String allClassVarNames.                                 "accumulated class variable names"
x := String sharedPools.                                      "immediate dictionaries used as shared pools"
x := String allSharedPools.                                   "accumulated dictionaries used as shared pools"
x := String selectors.                                        "message selectors for class"
x := String sourceCodeAt: #size.                              "source code for specified method"
x := String allInstances.                                     "collection of all instances of class"
x := String superclass.                                       "immediate superclass"
x := String allSuperclasses.                                  "accumulated superclasses"
x := String withAllSuperclasses.                              "receiver class and accumulated superclasses"
x := String subclasses.                                       "immediate subclasses"
x := String allSubclasses.                                    "accumulated subclasses"
```

```
x := String withAllSubclasses.                              "receiver class and accumulated subclasses"
b := String instSize.                                       "number of named instance variables"
b := String isFixed.                                        "true if no indexed instance variables"
b := String isVariable.                                     "true if has indexed instance variables"
b := String isPointers.                                     "true if index instance vars contain objects"
b := String isBits.                                         "true if index instance vars contain bytes/words"
b := String isBytes.                                        "true if index instance vars contain bytes"
b := String isWords.                                        true if index instance vars contain words"
Object withAllSubclasses size.                              "get total number of class entries"

"************************************************************************
 * debuging:                                                            *
 ************************************************************************"
| a b x |
x yourself.                                                 "returns receiver"
String browse.                                              "browse specified class"
x inspect.                                                  "open object inspector window"
x confirm: 'Is this correct?'.
x halt.                                                     "breakpoint to open debugger window"
x halt: 'Halt message'.
x notify: 'Notify text'.
x error: 'Error string'.                                    "open up error window with title"
x doesNotUnderstand: #cmrMessage.                           "flag message is not handled"
x shouldNotImplement.                                       "flag message should not be implemented"
x subclassResponsibility.                                   "flag message as abstract"
x errorImproperStore.                                       "flag an improper store into indexable object"
x errorNonIntegerIndex.                                     "flag only integers should be used as index"
x errorSubscriptBounds.                                     "flag subscript out of bounds"
x primitiveFailed.                                          "system primitive failed"

a := 'A1'. b := 'B2'. a become: b.                          "switch two objects"
Transcript show: a, b; cr.

"************************************************************************
 * Misc.                                                                *
 ************************************************************************"
| x |
"Smalltalk condenseChanges."                                "compress the change file"
x := FillInTheBlank request: 'Prompt Me'.                   "prompt user for input"
Utilities openCommandKeyHelp
```

Chris Rathman / *ChrisRath@aol.com*